FORTH-ICS / TR - 222 July 1998

Heap Management in Hardware

Ioannis Mavroidis

ABSTRACT

Critical time priority queues require hardware instead of software management. The heap data structure is an efficient way to organize a priority queue. In this report we present the various design issues and evaluate the hardware complexity and timing requirements of different possible hardware organizations of a heap using one FPGA and external memory. We also describe the organization and the datapaths we simulated in Verilog. Finally we present some more efficient techniques applicable only to ASIC implementations.

Heap Management in Hardware

Ioannis Mavroidis

Computer Architecture and VLSI Systems Division (CARV)
Institute of Computer Science (ICS)
Foundation for Research & Technology - Hellas (FORTH)
P.O. Box 1385, Science and Technology Park of Crete,
Heraklion, Crete, GR-711-10 GREECE
Tel: +30 81 391662, Fax: +30 81 391661
email: maurog@ics.forth.gr

Technical Report FORTH-ICS/TR-222 July 1998

Work performed under "ASICCOM" project and as a B.Sc. Thesis at the Univ. of Crete

ABSTRACT

Critical time priority queues require hardware instead of software management. The heap data structure is an efficient way to organize a priority queue. In this report we present the various design issues and evaluate the hardware complexity and timing requirements of different possible hardware organizations of a heap using one FPGA and external memory. We also describe the organization and the datapaths we simulated in Verilog. Finally we present some more efficient techniques applicable only to ASIC implementations.

This text is available in Postscript form, by anonymous ftp, from server "ftp.ics.forth.gr".
Directory: "tech-reports/1998".
Files: "1998.TR222.Heap_Management_in_Hardware.README",
"1998.TR222.Heap_Management_in_Hardware.ps.gz"

Contents

1 Introduction **1**

2 Operations supported by a heap **3**

2.1 Increment operation 3

2.2 Insert operation 4

2.3 Delete operation 5

3 Analysis **6**

3.1 Heap memory organization: 1 SRAM - horizontal 7

3.2 Heap memory organization: 2 SRAMs - horizontal 7

3.3 Heap memory organization: 1 SRAM - vertical 8

3.4 Heap memory organization: 2 SRAMs - vertical 8

3.5 Results - Conclusions 9

4 Simulation in Verilog **12**

4.1 Datapath for Increment 13

4.2 Datapath for Insert 14

5 Pipelining heap operations for an ASIC implementation **16**

5.1 Analysis of pipelined heap operations 17

6 Conclusion **19**

7 Acknowledgments **19**

Abstract

Critical time priority queues require hardware instead of software management. The heap data structure is an efficient way to organize a priority queue. In this report we present the various design issues and evaluate the hardware complexity and timing requirements of different possible hardware organizations of a heap using one FPGA and external memory. We also describe the organization and the datapaths we simulated in Verilog. Finally we present some more efficient techniques applicable only to ASIC implementations.

1 Introduction

A priority queue is a data structure that maintains a set of elements, each with two associated values called *id* and *priority*, that is able to find in an eficient way the id of the element with the highest priority (e.g. lowest *priority* value). This work was motivated by the need for high speed priority queues, arising in modern packet switched communication systems. Such systems try to provide different Quality of Service (QoS) to different connections, depending on what the customer needs and pays for. In ATM networks, cells from different Virtual Circuits (VC's), belonging to different service classes, interact with each other when they are multiplexed at the same output link of a switch. It is the responsibility of the output scheduler to decide when each cell will be forwarded to the output link, thus providing different QoS to different VC's. An idealized but non-realistic algorithm for such a scheduler, is the Generalized Processor Sharing (GPS) policy, described in [1]. Recently, many Packet Fair Queueing (PFQ) algorithms have been developed that approximate GPS, with different complexity-accuracy tradeoffs [2], [3], [4], [5]. All these algorithms define a timestamp (usually a virtual departure time) for each cell or service class and then use a priority queue to order their departures or service order respectively. This priority queue has to be in high speed since it should be able to operate over thousands of elements during a cell time.

A simple but less efficient way to organize a priority queue would be a linked list sorted by the priority values of the elements. Although a linked list can locate or remove the highest priority element in O(1) time, inserting a new element or decreasing of the priority of an existing one, takes O(N) time, where N is the number of elements. For large values of N, a linked list would be too slow.

A more efficient way to organize a priority queue is the heap data structure. In fact a sorted linked list is a special case of a heap. A heap is organized as a balanced tree, completely filled on all levels except possibly the lowest, which is filled from left up to a point. Each node of the tree corresponds to an element, and stores the id and priority values of that element. All nodes have the same number of children, which we call the *degree* of the heap. All heaps satisfy the *heap property*: Each node has a higher priority (i.e. lower priority value) than all its children. A heap can provide the element with the highest priority in O(1) time since, due to the heap property, this element will be stored at the root. To remove the highest priority node, insert a new one, or decrement the priority of an existing one we need $O(log_{degree}N)$ time.

Unfortunately, when we have thousands of elements, even the heap data structure, implemented in software, may not be fast enough. This is why a H/W implementation of the heap management may be very useful for critical time priority queues.

This report focuses on the H/W implementation using one FPGA and external memory, which, with low cost, can achieve a satisfactory speed for most systems. If we are willing to pay for the added design and implementation difficulties of going to on-chip implementations, we can operate the heap in a pipelined fashion and achieve a $O(degree)$ time (independent of N) per operation, with latency still in the order of $O(log_{degree}N)$. To accommodate pipelining, all heap operations should operate top-to-bottom as will be discussed in section 5.

The organization of the rest of this report is as follows.

In section 2 we define the heap operations that should be supported by the hardware implementation and show the way they operate.

In section 3 we present and evaluate the various design issues and possible hardware organizations of a heap using one FPGA and external memory. We discuss the different possible memory organizations and estimate the timing requirements of each.

In section 4 we describe the organization and the datapaths that were simulated in Verilog.

Finally, in section 5, we present the modifications for pipelining when the heap is maintained on-chip.

2 Operations supported by a heap

The definition of the operations that should be supported by a heap, is closely related to its possible use in the output scheduler of ATM switches. Each VC belongs to a service class. The VC's of the same service class are serviced Round-Robin and cells of the same VC are serviced in a First-Come-First-Served basis. The output scheduler can use a heap to order the service of the VC's assigning a priority number per VC. This heap should hold all the contending VC's, namely the ones that have buffered cells contending for the output link. At each cell time the first cell of the VC with the lowest priority number is forwarded to the output, and the priority number of this VC is incremented so as to give priority to other VC's over the one just serviced, in the subsequent cell times. The amount of this increment is VC specific and depends on the service class this VC belongs to. The bigger this increment amount is, the less this VC will be serviced. For example, if we want VC_1 to have double service rate than VC_2, then VC_1 should have an increment amount half of that of VC_2.

The operations supported by a heap, are the following:

- **INCREMENT(amount)** returns the id of the element with the lowest priority value (i.e. the id of the root node) and increments its priority value by amount (thus decreasing its priority). This operation, as we mentioned earlier, is performed by the output scheduler whenever it serves a VC, so as to decrease the priority of the just served VC.
- **INSERT(id, priority)** inserts a new element (id - priority pair) into the heap. This operation is performed by the output scheduler whenever a cell is received, that belongs to a VC which has no other buffered cells contending for the output link.
- **DELETE()** returns the id of the min element and deletes it. This operation is performed by the output scheduler whenever it forwards the last buffered cell of a VC. If this is the case, the VC should be deleted from the heap since it has no more buffered cells contending for the output link.

Next, we will describe how each of these operations is performed and provide some remarks useful to analyze their timing requirements.

2.1 Increment operation

INCREMENT(amount) returns the id of the element with the lowest priority value in O(1) time since this corresponds to the root node of the heap, and increments its priority value by amount. After this increment, the heap property may not hold since the root may now have a larger priority value than some of its children. If this happens, in order to maintain the heap property the root node should switch places with the node that has the lowest value among its children. This process of "falling down" should continue until the root node has come to a location where all its children have higher priority values than its own (or become a leaf, thus having no children).

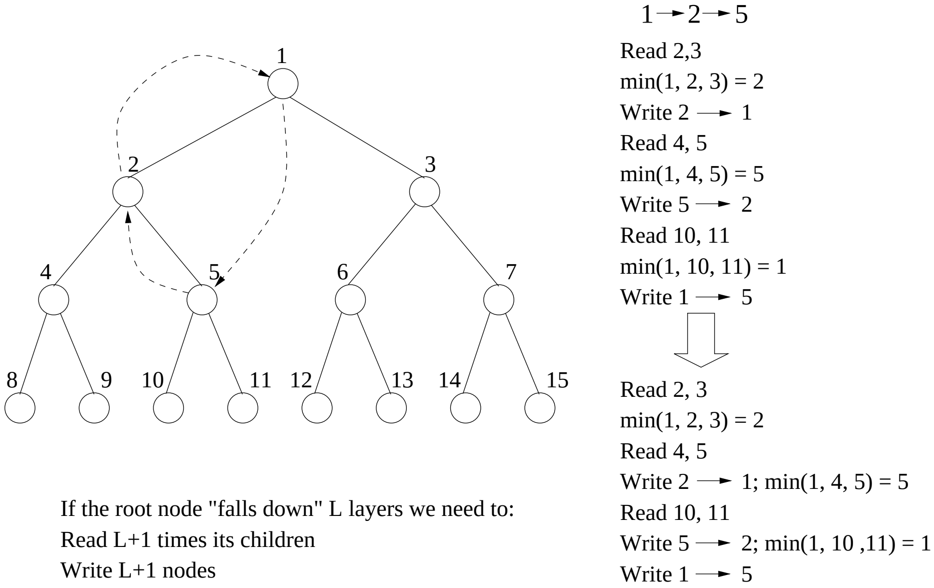


Figure 1: How INCREMENT operates

This is shown in Fig. 1. In this figure, we observe that the computation (i.e. comparisons) can be overlapped with memory accesses (except the first comparison), assuming that the memory bandwidth is not too high to be sustained by the hardware used for the computation (a case with too high bandwidth for FPGA hardware - but not for ASIC implementation - will be seen in 3.4). Thus, the time needed is determined by the number of memory accesses. We also see that if the root node falls down L levels, we need to read L+1 sets of its children nodes and write to L+1 nodes (in the example of Fig. 1 the root node fell down 2 levels, we read 3 times its children, and we wrote to 3 nodes).

2.2 Insert operation

INSERT(id, priority) inserts a new element (id - priority pair) into the heap. In order to maintain a balanced heap, the node corresponding to the new element is inserted at the leftmost empty location of the lowest level, or at the leftmost location of a new level if the lowest level is completely filled. This results in maintaining the heap balanced, but may destroy the heap property if the new node has a value less than its father. If this happens the new node should switch places with its father. Doing this will not violate the heap property among itself and its new children, since they all have higher values from their previous father which has a higher value from the new node i.e. their new father. This process of "going up" should continue until the new node has come to a location where its father has a lower priority value than its own (or become the root node, thus having no father).

This is shown in Fig. 2. As was the case with the INCREMENT operation, the comparisons (except the first one) can again be overlapped and the time needed is determined

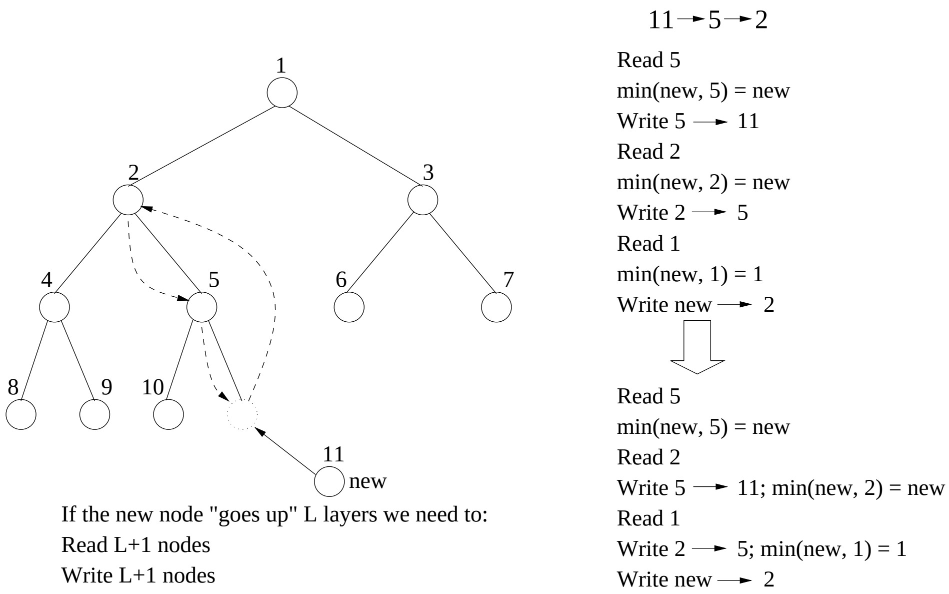


Figure 2: How INSERT operates

by the number of memory accesses. We also see that if the new node goes up L levels, we need to read L+1 nodes and write to L+1 nodes (in the example of Fig. 2 the new node went up 2 levels, we read 3 nodes and we wrote to 3 nodes).

A top to bottom operation of INSERT will be seen later in section 5. Since top to bottom INSERT has a higher latency than the above bottom to top INSERT, we assume that INSERT operates in a bottom to top way for an FPGA implementation (top to bottom is more suitable for ASIC implementation, as will be seen in section 5).

2.3 Delete operation

DELETE() reads the id of the root element and then deletes the element with the lowest priority value (i.e. the root node). This can be done if we overwrite the root node with the rightmost node of the lowest level and then start the process of "falling down" as we did in INCREMENT. This will maintain our heap balanced. Since DELETE is so similar to INCREMENT, we do not consider it separately in the rest of this report, and we will only analyze the INCREMENT and INSERT operations.

3 Analysis

For the memory organization of the heap management, several options had to be considered and their cost-performance trade-offs analyzed. These options are:

- The degree of the heap. Only powers of 2 were considered since this greatly facilitates the H/W organization of the heap management.
- The number of SRAMs used to store the heap. Only the cases of 1 and 2 SRAMs were considered due to cost constraints.
- The memory organization of each element. An element consists of two values, the id and the priority, each of which is considered to be 16 bits long. Thus, the heap will be able to hold $2^{16} = 65536$ nodes at most. The width of an SRAM was taken to be 32 bits. Two different memory organizations were considered: the *horizontal* and the *vertical*. In the horizontal organization each node is stored in a memory word (see Fig. 3a). In the vertical organization the nodes are stored two-by-two in two consecutive memory words (see Fig. 3b). The first word holds the id values and the second word holds the priority values, for the two nodes stored in them.

word k	id 1	prior. 1
word k+1	id 2	prior. 2

a) horizontal

word k	id 1	id 2
word k+1	prior. 1	prior. 2

b) vertical

Figure 3: Two possible memory organizations for the elements of the heap.

If we use vertical organization, we can read 2 priority values with one memory access. This can be exploited by INCREMENT which reads the priority values of the children in order to compare them with the priority value of the root node. Of course, the higher bandwidth provided by memory is only useful if the FPGA can support it; otherwise the vertical organization will degrade the performance since accessing a whole node (both its id and priority values) requires 2 memory accesses.

We will now analyze the time efficiency of the various combinations of the above options, in order to determine the time-cost trade-offs of each one, and decide which one we will simulate in H/W. The time efficiency is measured in clock cycles needed by each combination for the different actions of the heap. Remember that only INCREMENT and INSERT will be considered since DELETE is very similar to INCREMENT. We need only consider clock cycles used to access memory, since all other operations, except for exactly one comparison, can be overlapped with memory accesses as was previously observed. Each memory access requires one clock cycle. We will assume that the root node or the new node, "falls down" or "goes up", for INCREMENT and INSERT respectively, L number of levels. Since the different possible degrees of the heap have the same analysis for a given number of SRAMs and elements' memory organization, we will assume that the degree is equal to D, where D is a power of 2. Thus, we have the following combinations for a D-degree heap, with L-level operations: 1 SRAM - horizontal elements memory organization, 2 SRAMs - horizontal, 1 SRAM - vertical and 2 SRAMs - vertical.

3.1 Heap memory organization: 1 SRAM - horizontal

When the root node "falls down" L levels during INCREMENT, we recall from 2.1 that we need to read L+1 times its children nodes and write to L+1 nodes. Due to the horizontal memory organization for the elements, each node is stored in one word, and requires one memory access or clock cycle in order to read or write to it. Since the number of children nodes is equal to D, the degree of the heap, we have a total of $(\mathbf{D+1}) \times (\mathbf{L+1})$ clock cycles for INCREMENT.

During INSERT when the new node "goes up" L levels, we recall from 2.2 that we need to read L+1 nodes and write to L+1 nodes. Each read or write requires one memory access, giving us a total of $\mathbf{2} \times (\mathbf{L+1})$ clock cycles for INSERT.

3.2 Heap memory organization: 2 SRAMs - horizontal

If half of the children of every node reside in one SRAM and half in the other, then we can read them in parallel during INCREMENT requiring $\frac{D}{2} \times (L+1)$ memory accesses in each SRAM. Writes are serialized by the reads between them, requiring L+1 clock cycles. Thus, we have a total of $(\frac{\mathbf{D}}{\mathbf{2}} + \mathbf{1}) \times (\mathbf{L+1})$ clock cycles for INCREMENT.

The above partitioning of the nodes in the 2 SRAMs is the one that best exploits parallelism and provides maximum time efficiency for INCREMENT. This partitioning can be easily achieved if we use a breadth-first traversal of the heap to number the nodes, and use the least significant bit of the number of each node to identify the SRAM in which it is stored and the rest bits for the word address in this SRAM. One SRAM will then hold the even and one the odd children.

INSERT would better benefit from a partitioning of the heap into the 2 SRAMs where one SRAM holds the odd levels and the other SRAM holds the even. In this way, we will be able to "go up" in steps of two by reading the nodes of two adjacent levels two-by-two. Nevertheless, since INCREMENT is much more time consuming than INSERT, in an effort to minimize the worst case time needed by any operation, we choose to partition the heap in odd-even nodes.

The time required for INSERT is dependent on the path that the new node will follow as it "goes up". If all the nodes along the path are of the same parity (all even or all odd) they will all reside in the same SRAM. Then, the L+1 reads and the L+1 writes needed by INSERT will be serialized giving a worst case total of $\mathbf{2} \times (\mathbf{L+1})$ clock cycles (the same as when we used 1 SRAM). However, in average, half of the writes will be to a different parity node from the node that is to be read next, making it possible to parallelize these memory accesses. This results in an average total of $\mathbf{1.5} \times (\mathbf{L+1})$ clock cycles.

Note, that the two SRAMs are accessed in parallel only during INCREMENT and non-worst-case INSERT, and that they are accessed with the same address during INCREMENT. Thus, using one SRAM with 64 bits width instead, would give us the same worst case timing requirements for the heap operations with less cost and hardware complexity since the address bits would be almost halved.

3.3 Heap memory organization: 1 SRAM - vertical

At each step of the "falling down" during INCREMENT we read the children nodes in order to find the child with the minimum priority value which will switch places with the root node. When we have vertical memory organization for the elements of the heap (Fig. 3b) we need only read the priority values of the children, and not their id values, in order to find the child with the minimum priority value. If children nodes are put in consecutive memory locations, we can read their priority values two-by-two with $\frac{D}{2}$ memory accesses. This higher bandwidth is only useful if the FPGA can find the minimum of 3 priority values in one clock cycle [1]. Otherwise, vertical organization will add extra clock cycles as compared to horizontal, since in order to move the child node with the minimum priority value, we will have to read its id too (1 memory access) and write it to the position of its father (2 memory accesses since each node resides in two memory words). So, if we suppose that we can compare 3 priority values in one clock cycle, we have a total of $(\mathbf{\frac{D}{2}+3}) \times (\mathbf{L+1})$ clock cycles for INCREMENT.

Along the path followed during INSERT, every node except the last one will be read and written once (both its id and priority values). This requires 4 clock cycles for each such node. The last node will only be read partially since we do not need to read its id value. This saves one clock cycle. Thus, we have a total of $\mathbf{4 \times L + 3}$ clock cycles for INSERT.

3.4 Heap memory organization: 2 SRAMs - vertical

Like before, we will partition odd-even nodes in the 2 SRAMs, in order to exploit parallelism as much as possible when reading the children nodes during INCREMENT. In this way, each time, $\frac{D}{2}$ children will reside in each SRAM. The priority values of these children can be read four-by-four in $\frac{D}{4}$ clock cycles when we use vertical memory organization for the elements. Again, this higher bandwidth is only useful if the FPGA can find the minimum of 5 priority values in one clock cycle. Though this may not be feasible in many FPGA's (we assume a clock cycle of about 30ns), we will study this case, for the purpose of finding the latency of heap operations if an ASIC implementation is used (section 5). After finding the node with the minimum priority value, we need another 3 clock cycles in order to read its id value and write it to the memory location of its father. This results in $(\mathbf{\frac{D}{4}+3}) \times (\mathbf{L+1})$ clock cycles for INCREMENT.

[1] This can probably be achieved if we use 3 comparators to compare in parallel the 3 pairs formed by the 3 priority values, and use their results to extract the minimum priority value with one multiplexor.

The time required for INSERT is dependent on the path that the new node will follow as it "goes up". In the worst case all the nodes along the path will reside in the same SRAM giving a total time equal to the one we found using 1 SRAM, i.e. $\mathbf{4} \times \mathbf{L} + \mathbf{3}$ clock cycles. However, in average half of the writes will be to a different parity node from the node that is to be read next, making it possible to parallelize these memory accesses, saving us 2 clock cycles. This results in an average total of $\mathbf{3} \times (\mathbf{L} + \mathbf{1})$ clock cycles for INSERT.

Here too, as was the case with 2 SRAMs and horizontal memory organization for the elements of the heap, using one 64-bit width SRAM instead of two 32-bit width SRAMs, results in lower cost and hardware complexity while it has the same worst case timing requirements.

3.5 Results - Conclusions

We summarize the results from the above analysis in Fig. 4.

number of SRAMs	memory organization for elements: horizontal	memory organization for elements: vertical
1	increment: (D+1)*(L+1) insert : 2*(L+1)	increment: (D/2+3)*(L+1) insert : 4*L+3
2	increment: (D/2+1)*(L+1) insert : 2*(L+1) wc, 1.5*(L+1) ac	(Not for FPGA; ASIC only) increment: (D/4+3)*(L+1) insert : 4*L+3 wc, 3*(L+1) ac

Figure 4: Timing requirements of various memory organizations.

In Figures 5, 6, we show graphs of the cycles needed for operations INCREMENT and INSERT as a function of the elements currently in the heap.

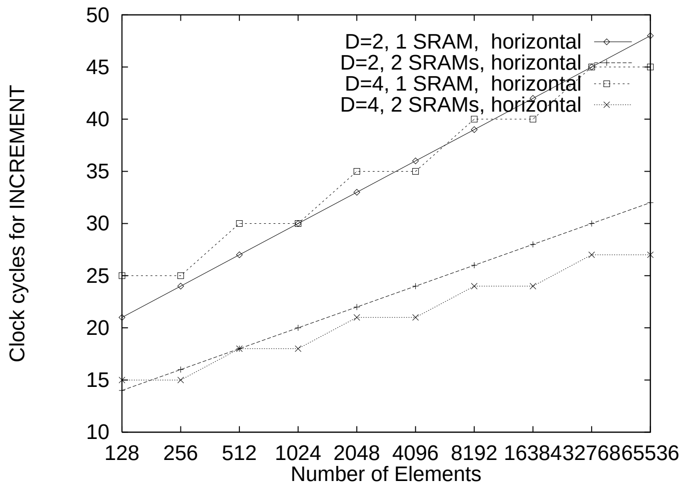


Figure 5: Timing requirements for INCREMENT.

To get a better idea of these results we applied different values for D and L and

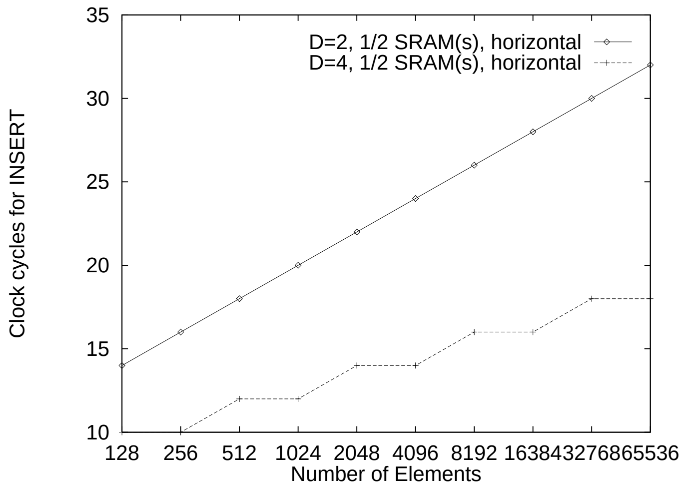


Figure 6: Timing requirements for INSERT.

obtained Fig. 7. The degree of the heap D was taken to be 2, 4, 8, or 16. For each degree, L was taken to be the depth of the heap when it holds 65536 elements, so that these values stand for the worst case timing requirements for INCREMENT and INSERT.

elem. mem. org.	horizontal								vertical							
degree	D = 2		D = 4		D = 8		D = 16		D = 2		D = 4		D = 8		D = 16	
levels	L = 15		L = 8		L = 6		L = 4		L = 15		L = 8		L = 6		L = 4	
number of SRAMs	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
increment	48	32	**45**	**27**	63	35	85	45	64	64	**45**	36	49	35	55	35
insert	32	32	**18**	**18**	14	14	10	10	63	63	**35**	35	27	27	19	19

Not for FPGA; ASIC only.

Figure 7: Clock cycles required for different mem. organizations.

The bold numbers correspond to the memory organizations with the minimum worst case timing requirements when 1 or 2 SRAMs are used.

Some observations drawn from the above two tables (Figures 4 and 7) are:

- As the degree D of the heap increases:
  - The depth of the heap decreases. The rate of this decrease is less than the rate of the increase of degree.
  - Worst case time for INCREMENT decreases and then keeps increasing. This is caused by the difference in the rates of change for the degree and depth of a heap.
  - Worst case time for INSERT decreases steadily.
- As the number of SRAMs increases:
  - Worst case time for INCREMENT decreases. This happens because the number of clock cycles required to read nodes decreases due to parallelism.
  - Worst case time for INSERT is constant. Average case time decreases.
  - The cost increases.
- Storing the elements in a vertical instead of a horizontal way, results in:
  - The number of clock cycles required to read priority values decrease, but the clock cycles required to read or write nodes increase. The proportion of clock cycles required to read priority values over the total clock cycles for INCREMENT increases with D, making it advantageous to use vertical instead of horizontal memory organization only for $D \geq 8$ (we try to decrease the worst case time for INCREMENT since it is much higher than that of INSERT).
  - The hardware complexity and cost increases if we want to exploit the higher bandwidth at which memory provides priority values. In order to be able to make comparisons with a higher bandwidth we need more comparators to operate in parallel.

4 Simulation in Verilog

As we see in Fig. 7 the minimum worst case timing requirements for 1 and 2 SRAMs are 45 and 27 clock cycles respectively. This difference can justify the higher cost and added implementation difficulties of using 2 SRAMs (or one with 64 bits width) only for highly time critical priority queues. In an effort to minimize the cost of the simulated H/W for the heap management, we decided to simulate one SRAM, and specifically the case with $D = 4$ and horizontal memory organization.

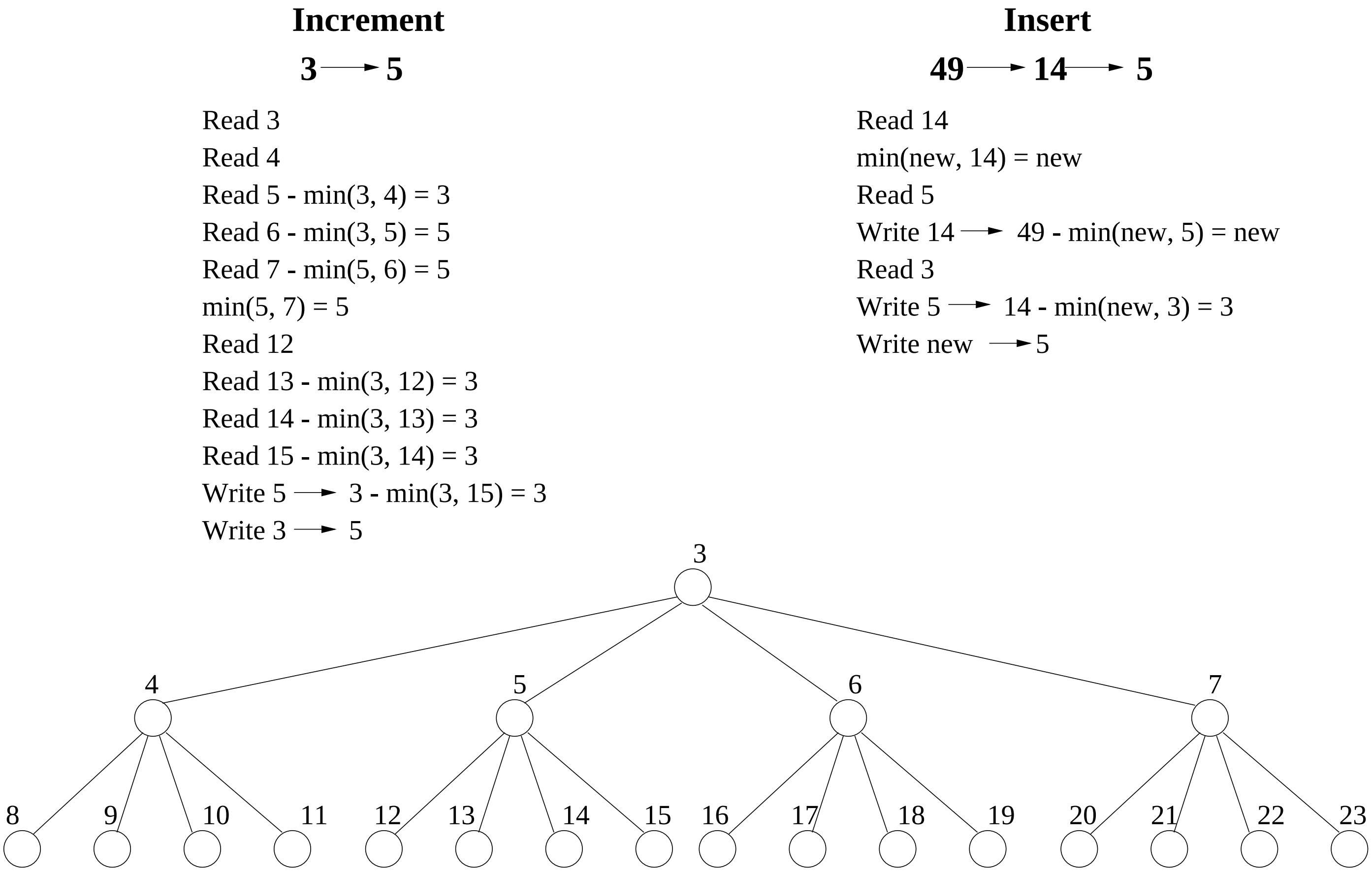


Figure 8: Examples of INCREMENT and INSERT operations for the simulated FPGA

The functionality of the simulated FPGA is shown in the examples of Fig. 8. The numbers near the nodes represent their memory locations. Using this numbering, the node stored at memory location N has its children stored in memory locations $4 \times N - 8$, $4 \times N - 7$, $4 \times N - 6$ and $4 \times N - 5$.

We will now briefly describe the datapaths needed to achieve the desired functionality for INCREMENT and INSERT operations. The datapath for DELETE is almost the same as the one for INCREMENT. The datapaths shown are somewhat simplified in the sense that, though they show all the different paths for the data, in an actual implementation some modifications would most probably be made in order to accommodate a small clock cycle (e.g. additional pipeline latches for tighter pipeline stages). These are the datapaths that were simulated in Verilog and tested using PERL.

4.1 Datapath for Increment

Fig. 9 shows a simplified datapath for INCREMENT operation.

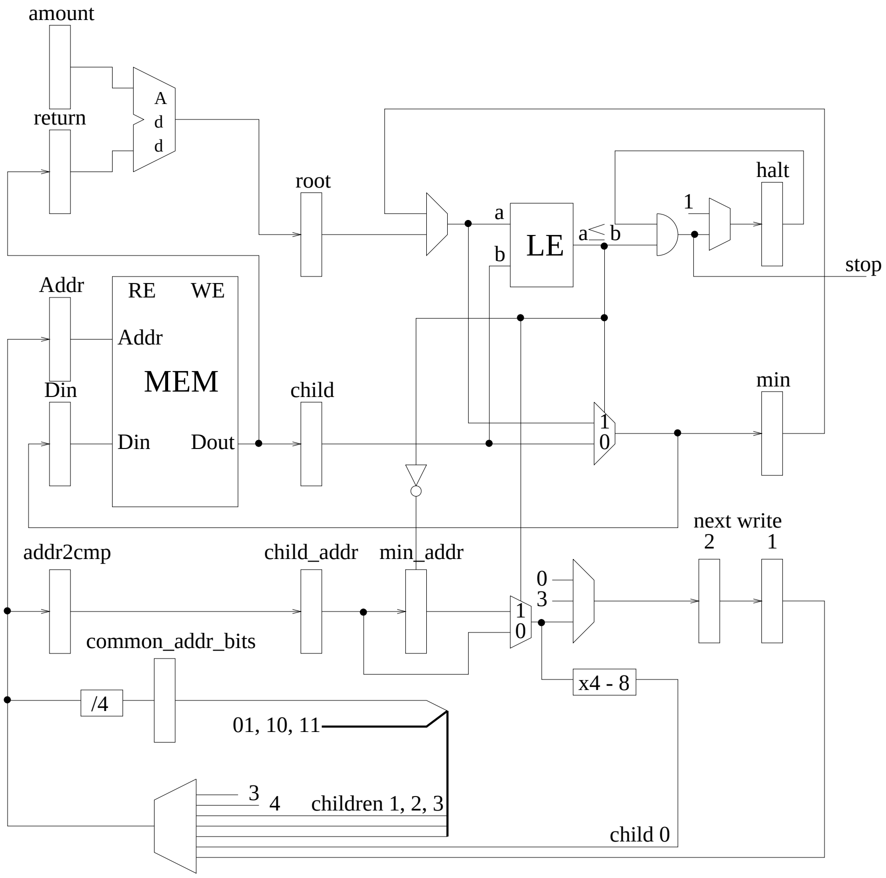


Figure 9: The datapath for INCREMENT operation

Register *root* holds the root node of the heap, after its priority value has been incremented by the amount specified by INCREMENT operation. The root node is also kept in register *return*, so that its id can be returned to the "caller" of INSERT. Register *child* holds one after another the 4 children nodes of root, at each step of its "falling down". In order to find the node with the minimum priority value among them, we make 4 comparisons, using the *LE* unit (less equal comparator), at 4 consecutive clock cycles (see example of Fig. 8). Register *min* holds the up-to-now minimum priority value node. The first child is compared with *root* and the next three children are compared one-by-one with *min*. Registers *child_addr* and *min_addr* hold the memory locations of the nodes stored in *child* and *min* registers respectively. The result of the last comparison, gives us the "minimum child", the node that should switch places with the root node, and determines 3 things (see example of Fig. 8 for the comparison min(5, 7) = 5):

- The node that we should read in the next clock cycle. This is the leftmost child of the "minimum child". The address of the "minimum child" is one of *child_addr* and *min_addr*, as selected by the outcome of *LE* in the last comparison. This address is multiplied by 4 and decreased by 8 in order to find the address of its leftmost child. The result is then sent to memory (register *Addr*).

- The node that is to be stored next. This is why the "minimum child" that results from the last comparison is stored in the *Din* register, and waits there to be used in the next "Write" instruction.
- The memory location that will be used for the "Write" after the following one (i.e. the second "Write" in the future). This is why we store the address of the "minimum child" in register *next write 2*. A "Write" always uses the address of register *next write 1*, and *next write 2* is shifted to *next write 1*. The two registers *next write 2, 1* are initialized with 3, 0 respectively.

Register *common_addr_bits* holds the common part of the memory locations of the 4 children. It is loaded when the memory location of the leftmost child is sent to register *Addr*, and is used afterwards to get the memory locations of the next 3 children by appending bits 01, 10 and 11 to it.

Finally, flip flop *halt* holds the logical AND of the 4 outcomes of *LE*. If this is 1, then the root node should stop "falling down" since it has come to a point where all its children have greater priority values than its own. To ensure that this also happens when it becomes a leaf, instead of checking whether the locations of its children are greater than N, we maintain the maximum positive numbers in the priority values of the children of all leaf nodes. To do this we initialize the memory (except memory location 2, see next section) with maximum priority values, and DELETE writes this maximum possible value to the priority value of the rightmost node of the last level (after moving it to the root of the heap, where it starts to "fall down").

4.2 Datapath for Insert

Fig. 10 shows a simplified datapath for INSERT operation.

Register *new* holds the new node to be inserted in the heap. Register *path_node* holds one after another the nodes along the path followed as the new node "goes up" during INSERT. Register *N* holds the number of nodes in the heap.

Registers *next write 2, 1* hold the memory locations of the next two "Writes". They are initialized to N, 0 respectively and whenever *next write 2* is shifted to *next write 1*, its new value comes from an increment by 8 and a subsequent division by 4 of the old value. This is the memory location of its father node. Notice in the example of Fig. 8 that the memory location we "Read" from, holds the father of the node that will be written by the next of the following "Write" (i.e. the second "Write" in the future). This is why whenever we "Read" we use the father of the address stored in register *next write 2*, and whenever we "Write" we use the address stored in register *next write 1*.

The next node to be stored is always the node with the greater priority value of the two being compared. We hold this node in register *Din* where it waits for the next "Write".

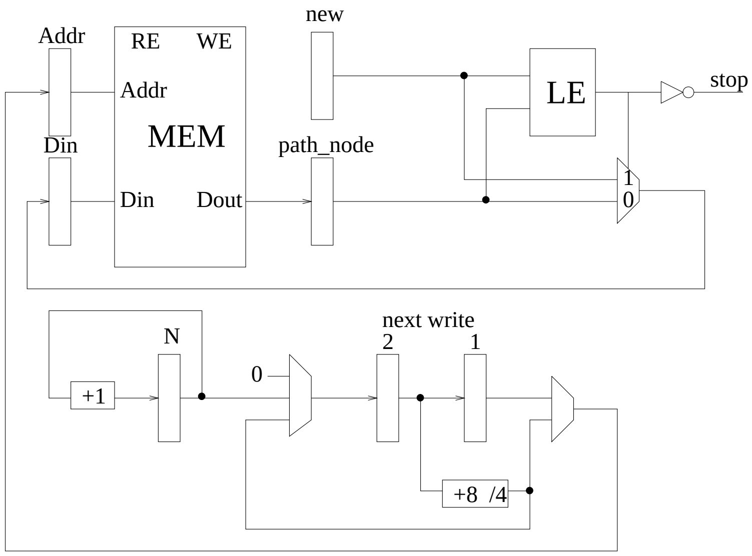


Figure 10: The datapath for INSERT operation

Finally, the process of "going up" should stop at the first comparison where the new node has greater priority value than its father node. When this happens, control signal *stop* is asserted. To ensure that this happens when the new node has come to the root of the heap, instead of checking whether its location is equal to 3, we maintain the lowest possible priority value in memory location 2. This works because 2 is seen as the address of the father of the root node, since $\frac{3+8}{4} = 2$.

5 Pipelining heap operations for an ASIC implementation

Another way for INSERT to operate is from top to bottom. This is illustrated in Fig. 11.

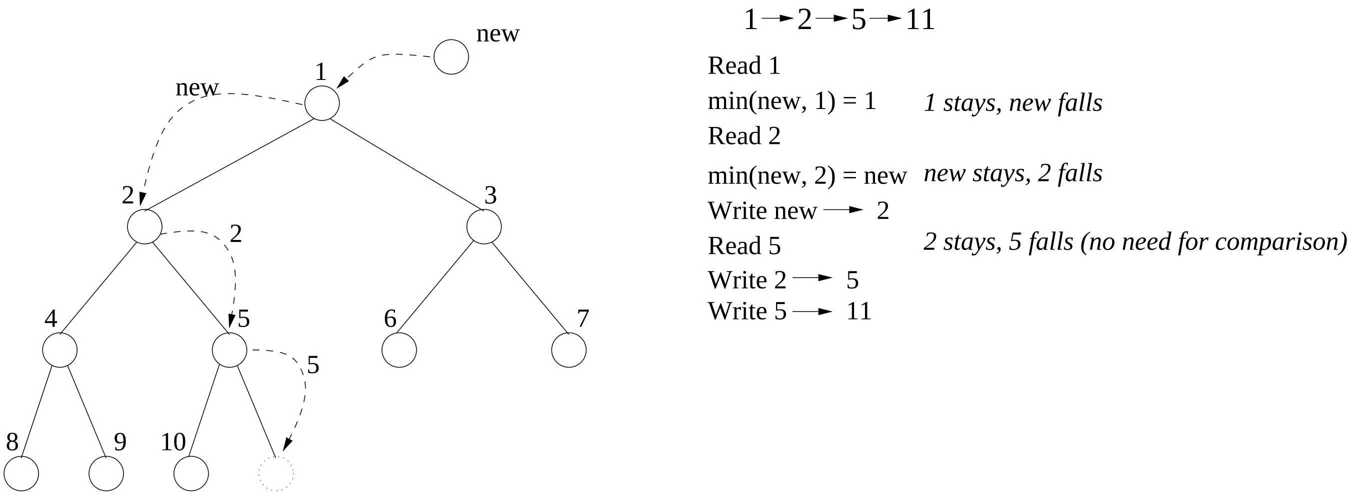


Figure 11: Top to bottom INSERT operation

The path followed during INSERT is from the root to the leftmost empty location of the lowest level (thus keeping the heap balanced). INSERT works in a recursive way. At each step the priority values of two nodes are compared. The node with the smaller one stays in the current position of the path and the other keeps "falling down" along the path (as though it is reinserted in the subtree rooted at the next node along the path).

If we implement INSERT in this top-to-bottom fashion, then all three heap operations will operate from top to bottom. This allows for simultaneous execution of different heap operations, each one operating in different levels of the heap than all others. Of course, this is only possible if all levels can be accessed in parallel, which is feasible if we use on chip memory instead of external SRAMs.

Two solutions are possible with different cost-applicability trade-offs:

- **Using on-FPGA memory.** Most FPGAs have on chip memory in the range of a few (usually less than 5) kilobytes. This memory is usually organized in a few memory blocks which can be accessed in parallel. For example, in ALTERA's FPGAs, memory is partitioned in 3 to 12 EABs (Embedded Array Blocks) each of which can be organized as 2048x1, 1024x2, 512x4 or 256x8. Unfortunately, the size of the on-FPGA memory is too small to hold more than a thousand nodes (e.g. ALTERA's FPGAs can hold no more than $\frac{12\times256}{4} = 768$ nodes). Since we are only interested in heaps with sizes of the order of tens of thousands nodes, we will not consider this case.
- **Using on-chip memory in an ASIC implementation.** Instead of using an FPGA for the hardware implementation of the heap, another more costly option is

to use VLSI design. Modern VLSI design makes it possible to have various memory blocks each one with its own size and read-write ports on the same chip. In such an implementation all levels can be accessed in parallel with one clock cycle to access a node.

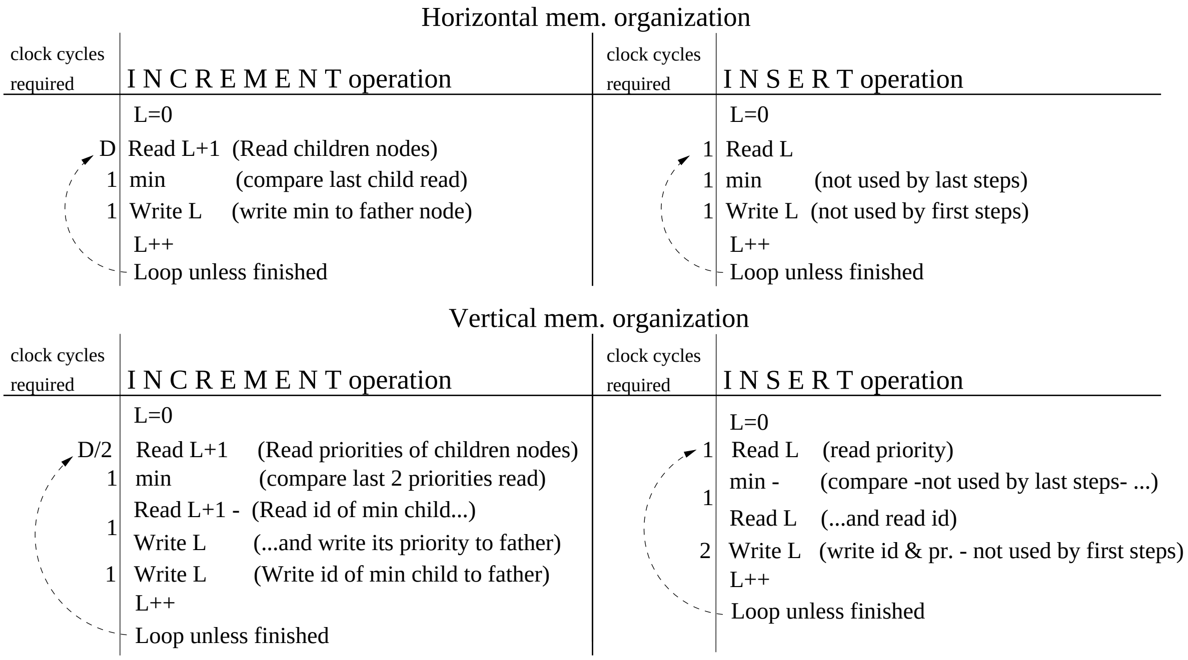
Since the first option above can not be applied to heaps with tens of thousands of nodes, we will adopt the second solution and suppose that all levels can be accessed in parallel with one clock cycle needed for each access.

5.1 Analysis of pipelined heap operations

Pipelining the heap operations requires that we deal with the various structural and data dependencies that arise. Since each on-chip memory block is identified by the heap level it holds, the dependencies that arise can be described as follows:

- **Structural dependencies.** Each level can only be accessed by at most one heap operation at each clock cycle.
- **Data dependencies.** No heap operation should access a level before all preceding operations have finished accessing this level.

Let's suppose that we have a D-degree heap. The operations of INCREMENT (Fig. 1) and INSERT (Fig. 11), give us the following timing diagrams. DELETE has to wait for at most D or $\frac{D}{2}$ clock cycles, for horizontal or vertical organization respectively, to read the rightmost node of the lowest level, and then its timing diagram will be the same as the one of INCREMENT.



In order to avoid the structural and data hazards a number of clock cycles should intervene between any two consecutive heap operations. We have to study 4 cases for each memory organization (number of clock cycles for vertical organization are put in square brackets):

- **INCREMENT after INCREMENT.** As we see in the timing diagram of INCREMENT for horizontal [vertical] mem. organization, it occupies two levels at every step of D+2 [vertical: $\frac{D}{2}+3$] clock cycles. If the first access of INCREMENT to a level is a Read at the beginning of step i then the last access to this level will be a Write at the end of step $i+1$. The difference in time of these two accesses is $2 \times D + 3$ [vertical: $D + 5$] clock cycles. Thus, in order to avoid a data hazard, a subsequent INCREMENT should start at least after $\mathbf{2 \times D + 4}$ [vertical: $\mathbf{D + 6}$] clock cycles.
- **INSERT after INCREMENT.** We see in the above timing diagrams that INSERT starts by Reading level 0 (root of the heap), while INCREMENT starts by Reading level 1. This is why a subsequent INSERT can start one step (3 clock cycles, 4 for vertical) earlier than a subsequent INCREMENT, i.e. at least $\mathbf{2 \times D + 1}$ [vertical: $\mathbf{D + 2}$] clock cycles after an INCREMENT.
- **INSERT after INSERT.** INSERT occupies each level for 3 (4 for vertical) clock cycles, by first reading and then writing to it. It is easy to see that in order to avoid a data hazard a subsequent INSERT should start at least after **3** [vertical: **4**] clock cycles.
- **INCREMENT after INSERT.** Since INCREMENT first accesses level 1 while INSERT accesses level 0, a subsequent INCREMENT should wait for INSERT to finish at least its first two steps. Thus, it should start at least after **6** [vertical: **8**] clock cycles.

From the above analysis we see that in order to maximize the sustainable throughput of heap operations we should minimize the degree of the heap (D). A value of 2 with horizontal memory organization would give the best performance (8 clock cycles per INCREMENT or INSERT operation), but would probably partition the on chip memory into too many memory blocks. If this is the case, we could opt for a value of 4 with vertical memory organization.

6 Conclusion

In this report we presented and evaluated the complexity and performance of various hardware organizations for the heap management. We focused on implementations using one FPGA and external memory, but we also investigated more efficient techniques applicable to ASIC implementations. By providing the datapaths for an FPGA implementation that we simulated, we showed that there exist simple, cost effective, and yet time efficient implementations for the heap management in hardware, applicable to critical time priority queues.

7 Acknowledgments

I would like to thank ICS-FORTH and "ASICCOM" project for funding my work, and professor Manolis Katevenis for his overall guidance and remarks.

References

[1] S. Keshav, *"An Engineering Approach to Computer Networking"* 1997 Addison-Wesley Professional Computing Series, ISBN 0-201-63442-2, "Generalized Processor Sharing" pages 234-237

[2] S. Keshav, *"An Engineering Approach to Computer Networking"* 1997 Addison-Wesley Professional Computing Series, ISBN 0-201-63442-2, "Weighted Fair Queueing" pages 238-252

[3] Jon C.R. Bennett and H. Zhang, *"WF^2Q: Worst-case fair weighted fair queueing."* In Proceedings of IEEE INFOCOM'96, pages 120-128, San Francisco, CA, March 1996.

[4] P. Goyal, H.M. Vin, and H. Chen. *"Start-time Fair Queueing: A scheduling algorithm for integrated services."* In Proceedings of the ACM-SIGCOMM 96, pages 157-168, Palo Alto, CA, August 1996.

[5] Jon C.R. Bennett and H. Zhang, *"Hierarchical Packet Fair Queueing Algorithms."* To appear in IEEE/ACM Transactions on Networking. Also in Proceedings of SIGCOMM'96, Aug, 1996.